**2. Add a Service Layer**

**The Benefits of Service Layer**:

Service Layer defines a common set of application operations available to different clients and coordinates the response in each operation. When we have an application that has more than one kind of client that consumes its business logic and has complex use cases involving multiple transactional resources - it makes sense to include a Service Layer with managed transactions.

With CRM, Sales and Inventory there will be a lot of CRUD-type use cases of which there is almost always a one-to-one correspondence with Service Layer operations. The responses to creation, update or deletion of a domain object should be coordinated and transacted atomically by Service Layer operations.

Another benefit of having a Service Layer is that it can be designed for local or remote invocation, or both - and gives us the flexibility to do so. The pattern lays the foundation for encapsulated implementation of an application's business logic and invocation of that logic by various clients in a consistent manner. This means we also reduce/remove duplication of code, as our clients share the same common services. We can potentially reduce maintenance costs too - as when our business logic changes, we (generally) only need to change the service, and not each of the clients.

**Service Layer Helps us**:

Service Layer helps us in the following areas.

1. Decouple your components
2. We can enforce specific business rules in your service layer which should be agnostic to your repository
3. Let a service facade one or more repositories. Let's consider the following sample

class Service {

private DatabaseBarRepo barRepo;

private DatabaseFooRepo fooRepo;

@Transactional

public void serviceRoutine() {

barRepo.doStuff();

fooRepo.doStuff();

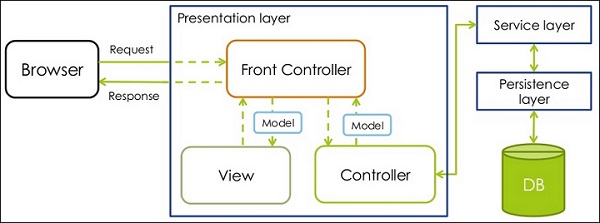
}

}

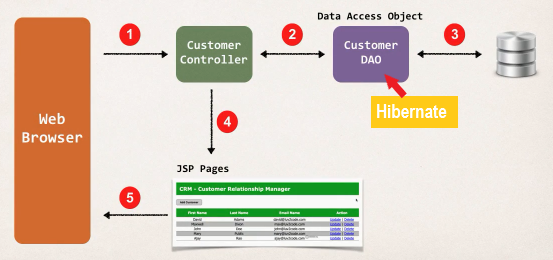
Here we let two separate repositories take part in the same transaction. This is specific for databases albeit the principles are valid for other systems as well.

**Layers of a Spring-MVC Application**:

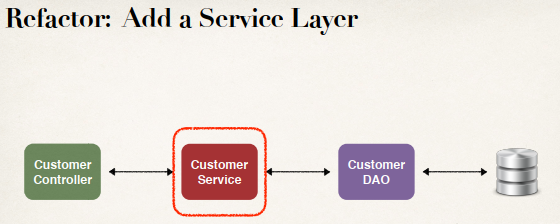
The following diagram represents the layers of a Spring-MVC application.



**Spring-MVC without Service Layer**:



**Add a Service Layer**:



**Purpose of Service Layer**:

* Service Facade design pattern
* Intermediate layer for custom business logic
* Integrate data from multiple sources (DAO/repositories)

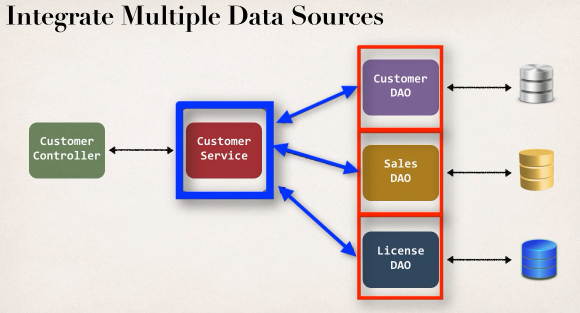
**Integrate Multiple Data Sources**:

Here is an example of Integrate data from multiple sources. For our customer service we may need to pull data about a customer from different types of locations. So up top we use the CustomerDAO, which gives us the basic information like firstName, lastName, email, address and so on.

We also may need to get a list of all of the sales products that they’ve purchased from our company. So, we will make use of our salesDAO.

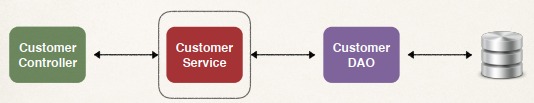
Finally, at the bottom we need to make use of the licenseDAO. Just so we can get a list of all of the software licenses that they have for our given products.

After that we will integrate all this together so that we can give the controller a single view of all that data that we’ve integrated and pulled together.



**Most Times - Delegate Calls**:

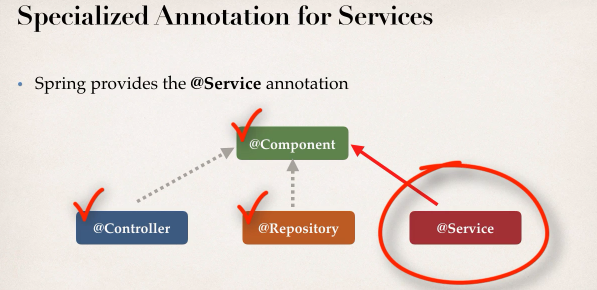
Now, most times when we start off, we have a very simple architecture. We have our Customer Service that simple delegates call to the customerDAO. This is a very good pattern to start with. This is best practice to create a Service layer and a DAO layer.



**Specialized Annotation for Services**:

Spring provide a specialized annotation **@Service** for the services. In previous we used **@Component**, **@Controller**, **@Repository**. Now we use **@Service**.

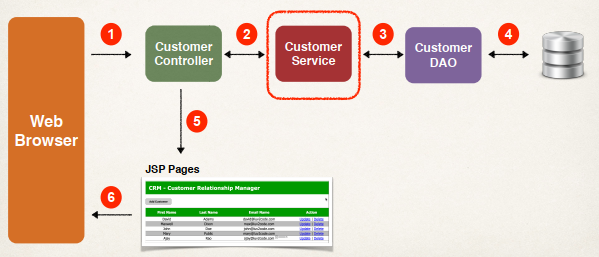
* **@Service** applied to Service implementations
* Spring will automatically register the Service implementation



**Development Process CustomerService (Step-by-Step)**:

1. Define Service interface
2. Define Service Implementation
3. Inject the CustomerDAO

**The Diagram**:



**1) Define Service Interface**:

**package** com.odduu.ruhul.service;

**import** java.util.List;

**import** com.odduu.ruhul.entity.Customer;

**public** **interface** CustomerService {

**public** List<Customer> getCustomers();

}

**2) Define Service Implementation**:

**package** com.odduu.ruhul.service;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** org.springframework.transaction.annotation.Transactional;

**import** com.odduu.ruhul.dao.CustomerDAO;

**import** com.odduu.ruhul.entity.Customer;

@Service

**public** **class** CustomerServiceImpl **implements** CustomerService {

@Override

**public** List<Customer> getCustomers() {

**return** customerDAO.getCustomers();

}

}

**3) Inject the CustomerDAO into CustomerServiceImpl**:

**package** com.odduu.ruhul.service;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** org.springframework.transaction.annotation.Transactional;

**import** com.odduu.ruhul.dao.CustomerDAO;

**import** com.odduu.ruhul.entity.Customer;

@Service

**public** **class** CustomerServiceImpl **implements** CustomerService {

//need to inject CustomerDAO

@Autowired

**private** CustomerDAO customerDAO;

@Override

@Transactional

**public** List<Customer> getCustomers() {

**return** customerDAO.getCustomers();

}

}

**Note**:

Here we remove **@Transactional** annotation from CustomerDAOImp and inject it to CustomerServiceImpl.

**Now we format our Customer controller**:

Inject CustomerService into CustomerController

@Controller

@RequestMapping("/customer")

**public** **class** CustomerController {

// need to inject our customer service

@Autowired

CustomerService customerService;

@GetMapping("/list")

**public** String listCustomer(Model theModel) {

...

**return** "list-customer";

}

}

2. Add a Service Layer